Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 149–160

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

MMC: the Mono Model Checker

Theo C. Ruys[1] Niels H.M. Aan de Brugh

Formal Methods and Tools group, Faculty of EEMCS,
University of Twente, Enschede, The Netherlands.

Abstract

The Mono Model Checker (MMC) is a software model checker for CIL bytecode programs. MMC has been developed on the MONO platform. MMC is able to detect deadlocks and assertion violations in CIL programs. The design of MMC is inspired by the Java PathFinder (JPF), a model checker for Java programs. The performance of MMC is comparable to JPF. This paper introduces MMC and presents its main architectural characteristics.

Keywords: software model checking, verification, MMC, MONO, .NET, CLI bytecode

1 Introduction

Software has become a normal part of our daily lives. People are working with software more than ever before, most of the times without even knowing it. The complexity of the design and implementation of a software system also has grown rapidly in the last decade. Televisions and mobile telephones are more complex and provide more functionality than a high-end workstation did less than a generation ago. Unfortunately, with the impressive advancement of technology and functionality comes unforeseen failure. The list of famous computer errors (better known as *bugs*) is long, and several had extreme consequences.

A mechanical and mathematically inspired technique that is specifically successful in finding errors in (functional designs of) software is called model checking. Model checking [4] is the *formal verification* of a *model* ($\mathcal{M}$) against a *specification* (p). The verification algorithm is an automatic and systematic process that formally concludes whether $\mathcal{M} \models p$ or $\mathcal{M} \not\models p$. Models are specified in abstract, high-level specification languages, e.g. PROMELA [12]. The property p is specified in some temporal logic, e.g. LTL. The idea is to first use a model checker to formally verify the design of a software system, and then afterwards, refine the (now correct)

[1] Corresponding author. Homepage: http://www.cs.utwente.nl/~ruys/

doi:10.1016/j.entcs.2007.02.066

verification model to a computer program. This original approach of applying a model checker to a high-level model $\mathcal{M}$ is referred to as *classic* model checking.

Although originally coined as a formal verification technique, the classic model checking approach proved more successful in finding bugs in systems than in actually proving that the systems were correct. Furthermore, due to the high-level constructs of the specification languages, the (mostly) manual refinement process of going from a high-level model to an implementation proved to be error prone. Over the last decade, however, model checking techniques have successfully been applied to software programs directly. Instead of dealing with a high-level model $\mathcal{M}$, the model checking tool is directly applied to a computer program.

This paper presents the Mono Model Checker (MMC), a software model checker for the verification of CIL bytecode programs. CIL stands for Common Intermediate Language and is the platform independent bytecode used within Microsoft's .NET. MMC has been implemented using the MONO development platform [21]. The MMC project has been initiated to get experience with the design and implementation of a software model checker. The core of MMC was intended to serve as a sandbox for further research on software model checking. Originally aimed as a proof-of-concept prototype, over time MMC developed into a competitive model checker for CIL bytecode programs.

An important advantage of CIL over Java bytecode is that CIL has been designed to be the target for many programming languages, not just C#. There exists compilers for imperative languages (e.g. Pascal, C), object-oriented languages (e.g. C#, Eiffel, Smalltalk), functional languages (e.g. Haskell, Lisp), script languages (e.g. Ruby, Lua, Perl) and even logic programming languages (e.g. Prolog). See [22] for a complete overview. This means that MMC is not just a model checker for C#, but can in principle be applied to programs written in many different programming languages.

Related work

Several approaches to *software* model checking can be distinguished.

- *Translation based checkers.* The source program is translated to the input language of some existing model checker. An example of this approach is the pair of tools: MODEX [13] and SPIN. MODEX (perhaps better known as its predecessor FEAVER) is a tool that can be used to mechanically extract high-level verification models from implementation level C code. These verification models can then be verified with SPIN. Another example of this approach is the first version of the Java PathFinder [10], that translated Java programs to PROMELA models, which were then verified with SPIN.
- *Abstraction based checkers.* An abstraction tool constructs an abstract (over-approximated) model of the original source program. This abstract model is subsequently analyzed. An example of this approach is SLAM [2], developed at Microsoft Research, a tool for reachability analysis of sequential C programs.
- *Bytecode checkers.* Bytecode checkers are built around the virtual machine of

some intermediate representation (in this context called bytecode). The effect of every intermediate instruction is analyzed by the virtual machine based checker. Such bytecode checkers indirectly analyze the complete full program: no abstractions are necessary.

MMC follows the bytecode based approach to sofware model checking. Below we briefly discuss some important software model checkers that also use this approach.

JPF. The Java PathFinder (JPF) [16] is a very successful software model checker for Java bytecode. It pioneered the concept of implementing a software model checker as a virtual machine that simulates the (binary) code of the application to be checked. JPF is an explicit-state model checker that systematically explores the state-space of a Java program, thereby generating it on-the-fly. It reduces the size of the state-space by applying partial order reduction techniques [8], as well as (heap) symmetry reduction [3,14]. The size of each individual state is reduced using the recursive indexing method [11]. By systematically exploring the state-space the JPF aims to find deadlocks and uncaught exceptions. JPF is an open source application and is available from [19].

XRT. At Microsoft Research, XRT [9], an exploration framework for .NET is being developed. XRT is a state exploration framework that follows similar goals as BOGOR [14] and JPF, using the approach of execution on the virtual machine level as pioneered by JPF. It supports the full safe (verifiable) CIL, and provides extension points on various levels, including the instruction set, the state representation, and the exploration strategy. It has been developed from the beginning together with one particular extension in mind, namely the unrestricted support of mixed concrete/symbolic state and exploration. Within XRT, instructions are represented by a language called XIL, which is an abstraction of CIL. Instead of the stack-based virtual machine of CIL, the virtual machine for XIL is a register machine. Although it means that a new virtual machine had to be developed, the use of a register language as an intermediate language opens new opportunities for optimizations. Instruction rewriters are being used to alter the code of methods. XRT is currently not publicly available.

BOGOR [14] is another successful software model checking approach. BOGOR accepts programs written in the BOGOR Input Representation (BIR) bytecode. The correlation between BIR and Java is very high, making BOGOR an obvious choice for the verification of object-oriented programs. There exist modules for BOGOR that perform symmetry reductions as well as partial-order reductions. Furthermore, BIR is an extensible language. It is possible to introduce new language features to extend BIR's syntax. More specifically, one can introduce new native types, and define operations on such types. The model checking framework of BOGOR is also extensible. An advantage of JPF and XRT over BOGOR is that they work directly (but *only*) on standardized bytecode formats: Java resp. CIL bytecode. BOGOR is much more of a model checking framework that can be used to build a sophisticated model checker than an off-the-shelf model checker like JPF and XRT. BOGOR is available from [17].

The development of MMC has been inspired by JPF. The architecture and design of MMC is also heavily based on JPF. And although the object-oriented design and the actual implementation of MMC (in C#) and organization of the classes and algorithms are different, all credits for the verification approach should go to the developers of JPF. Other than providing an actual model checker for CIL bytecode, we do not claim any major novelties here.

The next section describes the architecture of MMC and explains its most important components. Section 3 discusses some preliminary results and section 4 summarizes the paper and discusses some future directions for MMC.

2 MMC

This section describes the most important components of MMC. For a more detailed and thorough discussion on the design of MMC and its implementation, the reader is referred to [1]. Before describing MMC itself, we first discuss the context of .NET and the MONO project.

.NET

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. The specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. The intermediate language created by CLI-based systems is the Common Intermediate Language (CIL). CIL resembles an object oriented assembly language, and is entirely stack-based. It is executed by a virtual machine. Any language that can be compiled into CIL is called a .NET compliant language. Microsoft's implementation of the CLI is known as the Common Language Runtime (CLR), which defines an execution environment (i.e. a virtual machine) for the CIL code.

Mono

The MONO Project [5,21] is an open development initiative sponsored by Novell. MONO's objective is to develop an open source implementation of Microsoft .NET development platform. MONO offers a free and open implementation of the CLI published by ECMA as standard 335 [6]. MONO currently offers a number of components useful for developing new software: a run-time environment for CIL bytecode programs, compilers for C# and Visual Basic and an extensive class library, that can work with any CIL compatible language.

A MONO library that has proven to be extremely useful for the development of MMC is the CECIL library, developed by Jb Evain. It is not yet an official part of MONO, but can be downloaded separately [18]. CECIL is used within MMC to inspect so-called assemblies, which are compiled CIL bytecode fragments. Unlike the standard reflection framework in the class library of the CLI, CECIL is capable of inspecting those assemblies at the instruction level.

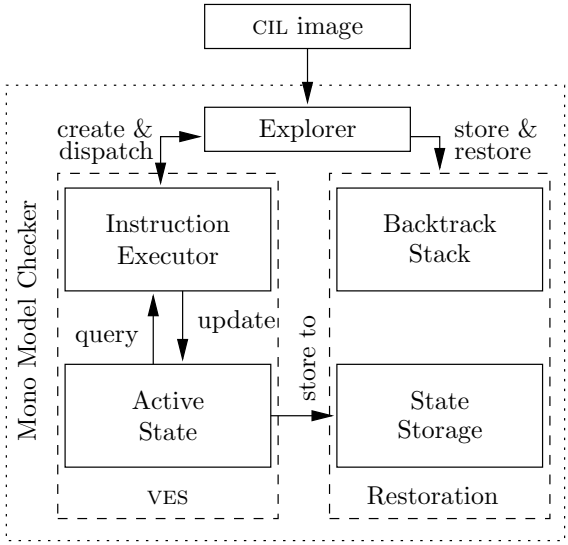

Fig. 1. MMC architecture

Design objectives of MMC

MMC is a software model checker for the verification of CIL bytecode programs. The intended audience for MMC is software developers who want to contribute to and experiment with a software model checker for .NET. Consequently, extensibility and simplicity of design has been a prime concern. Furthermore, readability and reusability of the implementation have been important objectives. The current implementation can be regarded as a stable but open 'sandbox' that can be used to experiment with new ideas and methods.

Approach

MONO applications – CIL bytecode programs – are run on a virtual execution system (VES). We have adopted the concept of implementing a model checking virtual machine (VM) pioneered by JPF, capable of systematically exploring the state space of a software application. The exploration is performed by iteratively executing instructions that are read from the compiled CIL bytecode. To have full control over this process, we manage all VM structures ourselves. The client application does not see any difference between running on MONO or on MMC, i.e. MMC behaves exactly the same as the MONO VES would. This is made easier by a large extend by the fact that MMC itself runs on the MONO VES. Calls and operations can be 'passed down' by MMC by performing exactly the same action as the client application.

Architecture

Figure 1 gives a schematic overview of the architecture of MMC, its most important building blocks and interaction between those blocks. Central to the architecture is the *Explorer*. This component drives the state space exploration. On the left-hand side in the figure, the part of the MMC that provides the virtual execution environment is depicted. It is based on two components: an active state and instruction

executors. The first holds the state of the virtual machine, i.e. the running threads, allocated objects, etc. The latter part of the VES, the instruction executors, are responsible for executing a single CIL instruction. A typical instruction executor queries and updates the active state. The construction and starting of the executors is done by the explorer. To the right we find the parts of the MMC that are used for storing and restoring states: the state storage and a backtrack stack. At certain points in the execution path, the explorer has to store the active state in the *state storage*. This storage is used to check if a certain state has already been explored in the past, so we do not need to explore it again. The *backtrack stack* contains the sequence of states (i.e. the *path*) that has been explored so far. It allows the explorer to restore a previously visited state. It contains scheduling related information, most importantly the threads that have not yet been run from that state, and all data needed to restore the previous state.

Explorer

As stated previously, the explorer's task is to systematically drive the exploration algorithm, which involves executing instructions, storing and restoring the state, and checking for safety properties. The explorer uses a depth-first-search (DFS) strategy to visit all reachable states. To detect cycles in the exploration graph, visited states are stored, and each new state is compared to all stored ones. Backtracking is done by keeping a stack of the states that form the current path being explored.

During exploration, MMC will check for deadlocks and assertion violations. A deadlock is a state where there are no runnable processes in the system, but not all processes have terminated. An assertion is a user-defined condition that has to hold in a certain state.

The instruction executors (IEs) are objects responsible for executing the CIL instruction. There are many IEs: one for each type of CIL instruction. Each IE is implemented by its own C# class. This approach closely resembles the *command* design pattern [7]. The merit of using the command pattern in the MMC is that code to execute CIL-instructions can be seen as first-class-citizens. This allows us to add more CIL-instructions to MMC without modifying existing code. Furthermore, meta-data (such as the list of exceptions an instruction can throw, the safety of instructions, etc.) that is associated with the instructions can be added to the executor classes, thereby eliminating the need for big look-up structures.

In MMC, a transition between two states consists of zero or more *safe* instructions together with one *unsafe* instruction. A *safe* instruction is an instruction whose execution is not visible (or relevant) to any of the other threads. An *unsafe* instruction is an instruction that *might* influence other processes. The *merging* of safe instructions into one transaction can be considered a mild form of partial order reduction [8]. SPIN employs a similar technique called statement merging [12]. Note that within MMC instruction merging is not a feature, but rather a necessity. Without instruction merging the number of states would soon become unmanageable.

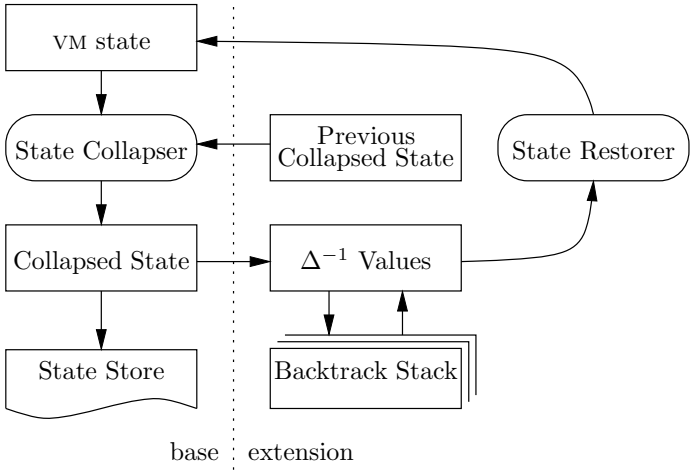

Fig. 2. State Storage.

State storage

All states that are visited by the explorer are stored. However, MMC does not only suffer from the infamous state explosion, the individual VM states can become quite big as well. To reduce the size of the states that are being stored, MMC uses a technique called *recursive indexing* (or *collapse compression*), which is also used in SPIN [11] and JPF.

The principle of recursive indexing is as follows. There is a data structure called *pool* that stores objects. Each stored object (O) is assigned a *unique* indexing number by the pool ($\mathcal{P}$). That is, the indexing number of two objects is the same if and only if the two objects are the same. Once an object is stored, it is never removed, and once assigned, the index number remains the same. Assume we have a part of the state that consists of several objects in a fixed order: $L_1 = [O_1, O_2, \ldots O_n]$. We store these n object in pool $\mathcal{P}$, and replace all objects by their respective index numbers in $\mathcal{P}$, giving us the list of numbers $C_1 = [\mathcal{P}(O_1), \mathcal{P}(O_2), \ldots, \mathcal{P}(O_n)]$. We call this translation *collapsing*. There is no loss of identity, since every object has a unique number. Figure 2 shows the state storage organization of MMC. The left-hand side of the figure shows the 'base' scenario which uses the collapsing method.

Although the collapse method is quite good at compressing the state there is still an aspect that is not optimal. Every time we have to check whether we have seen a state, the state first has to be collapsed in order to compare it to other states seen. This is mostly redundant work, because two consecutive states S_1 and S_2 do not differ that much. In other words, many of the objects that S_1 and S_2 consists of are the same. The solution to this problem is straightforward. We keep a copy of the previous collapsed state S_1 and only collapse the parts of S_2 that differ from S_1, i.e. those that have actually changed. The right-hand side of Fig. 2 shows this extension of the data flow. Note that although conceptually simple, this involves keeping track of all changes in the active state as code is being executed by MMC's virtual machine.

This *delta* between collapsed states proved also useful and effective in other

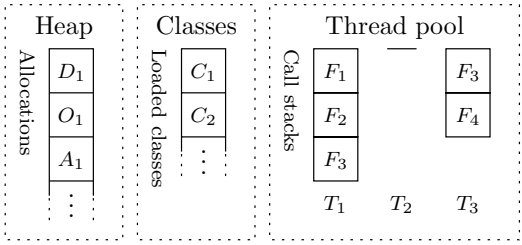

Fig. 3. Active state of the virtual machine.

parts of the explorer. Because MMC uses a DFS strategy to visit all reachable states, it has to *backtrack* when it encounters a state it has already visited. Instead of storing all (collapsed) states on the backtrack stack, MMC follows SPIN's approach and stores the reverse transition between two consecutive states on the stack. Given a state S_2, this reverse transition can be used to rebuild the previous state S_1. This reverse transition is called *reverse delta*, depicted by Δ^{-1}. The right-hand side of Fig. 2 shows how the Δ^{-1} values are computed and used by the explorer when backtracking.

Active state

An important module of MMC is the *active state* (see Fig. 1). The active state holds the current state of the VES of MMC. The active state is being queried and updated by the instruction executors, and it can be stored in the state storage. Although at a design level the active state is not that hard to grasp, it is its size and complexity that gave us a hard time managing the structure at run-time in an efficient way.

Fig. 3 shows an example of an active state. Three components can be identified, i.e. the heap, classes and thread pool. The *heap* holds all dynamic allocations: objects, arrays and delegates. The static fields are stored in the *classes* component. The *thread pool* contains the concurrent processes that are currently running. For each process it contains the complete stack of method calls by that process.

Heap

The heap is the part of the active state where dynamic allocations are stored. It is used often, since many modern (object-oriented) programming languages use dynamic allocations (objects) for everything more complex than a simple number.

MMC's explorer needs to be able to check two heaps for equivalence. This is a bit cumbersome due to the highly dynamic nature of the heap. As a solution, MMC uses the same heap symmetry reduction technique as implemented in both JPF and XRT: when a new allocation is created on the heap for the first time, we remember where it has been put. The next time we create the *same* allocation, we put the object in the same place. In this way, the order in which the allocations are created is not longer relevant and always yields the same heap.

Garbage collection

Memory deallocation in CLI is automatic: the programmer does not have to explicitly specify which objects to deallocate. The garbage collector of the VES (i) determines what data objects in a program will not be accessed in the future and (ii) reclaim the storage used by those objects.

Garbage collection is typically an expensive algorithm; it normally is only started by the virtual machine when the free space on the heap is running out. Within MMC this is not acceptable though. Because of the state matching algorithm that is being executed by MMC explorer, we cannot allow 'dead' objects to stay in the heap. For this reason, after each transition MMC calls its own garbage collecting algorithm to check which objects in the heap are dead. Only the live objects are retained in the active state.

Two separate garbage collection algorithms are implemented in MMC, i.e. reference counting and mark and sweep. Note that reference counting is redundant when mark and sweep is applied, and is only available as a faster but less rigorous technique. An allocation is garbage if no reference to it exists anywhere in the active state. The *reference counting* mechanism is (as the name suggests) based on the idea of keeping track of how many references point to an allocation. If this number reaches zero, the allocation can be removed. An important drawback of reference counting is that it cannot deal with cyclic structures. The mark and sweep algorithm consists of two phases. In the *mark*-phase, all allocations that are reachable are marked. In the *sweep*-phase, all allocations that have not been marked are deleted. The time complexity of the mark and sweep algorithm is somewhat higher than that of reference counting.

Current status

The current version of MMC is version 0.5.1. The development of MMC 0.5.1 took roughly one man year of work. The code base of MMC consists of 16k lines of C# code and constitutes 460Kb of source code (including doxygen documentation). A binary version of MMC 0.5.1 is currently available from [20]. We are planning to release an open source version of MMC in the beginning of 2007.

3 Experiments

We have performed some preliminary experiments with MMC on some C# programs. To get a feeling of the performance of MMC, we have translated these examples also to JAVA, and checked these with JPF. The experiments have been carried out on an AMD Thunderbird 700 CPU with 768Mb of RAM running Linux. We used MONO version 1.1.13.6. In this paper, we only briefly discuss the results on the well-known dining philosophers problem.

As both MMC and JPF stop exploring the states space as soon as a deadlock has been found, the number of states visited is highly depended on the scheduling algorithm used. MMC scheduler does not consider fairness at the moment, which gives MMC very good results on this particular test. We stress that the comparison

	number of threads							
	2		**3**		**5**		**10**	
	#	sec	#	sec	#	sec	#	sec
MMC	65	0.35	37	0.39	182	0.44	372	0.70
MMC w/o sharing	26	0.34	106	0.35	59	0.37	114	0.46
JPF	20	0.84	64	1.11	376	2.3	14204	82

	12		**20**		**50**		**80**	
	#	sec	#	sec	#	sec	#	sec
MMC	448	0.90	752	1.53	1892	7.7	3032	19
MMC w/o sharing	136	0.50	224	0.85	554	2.7	884	6.4

Table 1
Number of states and run-time (in seconds) for dining philosophers example.

with JPF is not fair (w.r.t. the number of states), but are only included here to give an impression of the performance of MMC.

Table 1 shows the results of this experiment. The data is presented as 'number of states / time in seconds'. The first row shows the results of MMC and the last row the results of JPF. We also measured what happens when we consider all accesses to the heap to be safe. That is, we assume no objects are shared (i.e. MMC w/o sharing). This is a very optimistic setting, but it is safe for this example. Not surprisingly, it results in a serious reduction in run-time and the number of states.

We investigated also how MMC performs when it does not stop when a deadlock is found. Note that these measurements are not comparable with JPF, since JPF stops when it finds a deadlock. For a system of 3 philosophers, and with sharing disabled it took MMC 29 seconds to explore a complete state space of 11564 states. We tried a system of 4 philosophers, but aborted the exploration after 23 minutes and visiting more than 220500 states. For this last example, the ratio for MMC is roughly 160 states per second. From Table 1 we learn that for 10 threads, JPF visits roughly 170 states per second.

Table 1 also shows that the ratio states/sec is not constant. When the number of philosopher threads grows, this ratio goes down. This is not surprising as several algorithms in MMC (and JPF) are dependent on the size of the state vector (i.e. collapsing, garbage collection, state matching, etc.) So when the size of the states grows, the relative performance of MMC (and JPF) will go down.

4 Conclusions

In this paper we presented MMC, a model-checker for CIL bytecode programs. Due to several refactorings, the design and implementation of MMC is clear, readable and extensible. We feel that MMC is a useful platform in an academic environment where ease of experimentation with different implementations is an important virtue.

Future work

After several high-level optimizations to reduce both the number of states and the size of these states, the speed of MMC proved to be comparable with JPF: around

150 states/sec. The current version of MMC can already be used to verify small C# programs. To extend the usability of MMC further, several improvements are planned.

- MMC's classification of instructions into safe- and unsafe instructions can been regarded as a limited form of partial order reduction. Although already effective, much more can be won here if we would take into account the sharing of objects between the different threads. JPF, for instance, uses the mark phase during garbage collection to analyze the objects that are being shared.
- XRT and JPF support symbolic states and symbolic evaluation. This makes both tools capable of analyzing much larger programs than using explicit states alone. Symbolic extensions to MMC are planned as well.
- The heap symmetry reduction algorithm currently implemented in MMC is quite effective, but not optimal. Further reductions might be achieved by implementing the kBOTS-algorithm as implemented in BOGOR [15].
- When MMC finds a deadlock or assertion violation, a rather concise and cryptic backtrack stack is presented to the user. To allow more detailed, user-readable error traces, we are planning to adopt the DWARF debugging standard such that the error trace can be replayed in DWARF-compatible debuggers like gdb.
- The current version of MMC has only been tested with C# programs. However, one of the objectives of the MMC project (and benefits of the CIL framework) is to make the model checking framework source language independent.

References

[1] Aan de Brugh, N. H. M., "Software Model Checking for MONO," Master's thesis, University of Twente, Enschede, The Netherlands (2006).

[2] Ball, T. and S. K. Rajamani, *The SLAM Project: Debugging System Software Via Static Analysis*, in: *Proc. of the 29th Symp. on Principles of Programming Languages (POPL 2002)*, 2002, pp. 1–3.

[3] Clarke, E. M., E. A. Emerson, S. Jha and A. P. Sistla, *Symmetry Reductions in Model Checking*, in: A. J. Hu and M. Y. Vardi, editors, *Proc. of the 10th Int. Conf. on Computer Aided Verification (CAV 1998)*, LNCS **1427** (1998), pp. 147–158.

[4] Clarke, E. M., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[5] Dumbill, E. and N. M. Bornstein, "Mono: A Developer's Notebook," O'Reilly, 2004.

[6] ECMA International, *Standard ECMA-335: Common Language Infrastructure (CLI)* (2005), http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[7] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[8] Godefroid, P., "Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem," PhD in Computer Science, University of Liege (1994), A revised version has been published as LNCS 1032, Springer-Verlag (1996).

[9] Grieskamp, W., N. Tillmann and W. Schulte, *XRT: Exploring Runtime for .NET - Architecture and Applications*, Electr. Notes Theor. Comput. Sci. (ENTCS) **144** (2006), pp. 3–26.

[10] Havelund, K. and T. Pressburger, *Model Checking JAVA Programs Using JAVA PathFinder*, International Journal on Software Tools for Technology Transfer (STTT) **2** (2000), pp. 366–381.

[11] Holzmann, G. J., *State Compression in* SPIN*: Recursive Indexing and Compression Training Runs*, in: *Proc. of the 3th International SPIN Workshop, University of Twente, Enschede, The Netherlands*, 1997.

[12] Holzmann, G. J., "The SPIN Model Checker – Primer and Reference Manual," Addison-Wesley, Boston, Massachusetts, USA, 2004.

[13] Holzmann, G. J. and M. H. Smith, *Software Model Checking*, in: J. Wu, S. T. Chanson and Q. Gao, editors, *Proc. of FORTE/PSTV 1999*, IFIP Conference Proceedings **156** (1999), pp. 481–497.

[14] Robby, M. B. Dwyer and J. Hatcliff, BOGOR*: An Extensible and Highly-modular Software Model Checking Framework*, in: *Proc. of ESEC/SIGSOFT FSE* (2003), pp. 267–276.

[15] Robby, M. B. Dwyer, J. Hatcliff and R. Iosif, *Space-Reduction Strategies for Model Checking Dynamic Software*, Electr. Notes Theor. Comput. Sci. (ENTCS) **89** (2003).

[16] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model Checking Programs*, Automated Software Engeneering **10** (2003), pp. 203–232.

[17] *Bogor*, http://bogor.projects.cis.ksu.edu/.

[18] *Cecil*, http://www.mono-project.com/Cecil.

[19] *Java PathFinder*, http://javapathfinder.sourceforge.net/.

[20] *MMC: The Mono Model Checker*, http://www.cs.utwente.nl/~ruys/mmc/.

[21] *The Mono Project*, http://www.mono-project.com.

[22] *.NET Languages*, http://www.dotnetlanguages.net/.